

20. The system of claim 19, wherein said appropriate input comprises a variable having a data type which is assignment compatible for the expression.

Remarks

Status of application

Claims 1-20 are pending in the subject application. Applicant gratefully acknowledges the Examiner's indication of allowable subject matter in claims 4, 7, 12-14 and 19-20 (subject to rewriting those claims in independent form including base and intervening limitations). Other claims stand rejected in view of prior art. By this Amendment, Applicant has amended certain claims in an effort to better distinguish the claimed invention. Re-examination and reconsideration of the claims, as amended, are respectfully requested.

General

A. Drawing change

The Examiner has indicated that Figs. 1A, 1B, and 2 should be labeled as "Prior Art." A Request for Drawing Change is filed herewith, for purposes of entering the Examiner-specified changes. However, it should be understood that the Visual Development System 200 (which appears in Fig. 1B) can only be considered prior art to the extent that it is exclusive of the Code Completion invention described by Applicant.

B. Allowable subject matter

The Examiner has indicated allowable subject matter in claims 4, 7, 12-14 and 19-20, subject to rewriting those claims in independent form to include all of the limitations of the base claim and any intervening claim. The claims have been amended accordingly and, thus, are believed to be in condition for immediate allowance.

C. Reconsideration of claims 5 and 6

Dependent claims 5 and 6 depend from claim 4, which the Examiner has indicated recites allowable subject matter. Claim 4, as mentioned above, has been amended to include the limitations of base claim 1; there are no intervening claims. As claim 4 recites

limitations which the Examiner has indicated distinguish over the art, it is respectfully requested that the Examiner reconsider the rejection of claims 5 and 6. In particular, claims 5 and 6 add further limitations beyond those limitations recited in their immediate parent, allowable claim 4.

The invention

A visual development system of the present invention provides an interface that includes an Integrated Development Environment (IDE) interface having a code editor. The IDE provides "Code Insight" functionality to the code editor for displaying context sensitive pop-up windows within a source code file. Of particular interest to the present invention are Code Completion and Code Parameter features of Code Insight.

Code Completion is implemented at the user interface level by displaying a Code Completion dialog box after the user enters a record or class name followed by a period. For a class, the dialog lists the properties, methods and events appropriate to the class. For a record or structure, the dialog lists the data members of the record. To complete entry of the expression, the user need only select an item from the dialog list, whereupon the system automatically enters the selected item in the code.

Code Completion also operates during input of assignment statements. When the user enters an assignment statement for a variable and presses a hot key (e.g., `<ctrl><space_bar>`), a list of arguments valid for the variable is displayed. Here, the user can simply select an argument to be entered in the code. Additionally, the user can select a type which itself is not appropriate (e.g., *record* type) but nevertheless includes a nested data member having a type which is valid. In an integer assignment statement, for example, the user can select a type variable *SMTP1*, a structure of type *TSMTP* which contains an integer data member. Upon the user entering the dot operator after *SMTP1*, the system displays a list of valid data members for *SMTP1*, that is, the data members which are assignment compatible for the integer assignment. Now, the user can simply select a valid member to be entered in the code.

Similarly, the user can bring up a list of arguments when typing a procedure, function, or method call and needs to add an argument. Consider, for instance, a scenario where the user has begun entry of a *SendFile* method call. The *SendFile* method itself is defined elsewhere in the code as follows.

```
procedure SendFile(Filename: string);
```

Upon the user entering the opening parenthesis, the system automatically displays parameter information for the call. In this manner, the user can view the required arguments for a method as he or she enters a method, function, or procedure call.

Prior art rejection: 35 USC Section 102(b): Request for reconsideration

Claims 1-3, 5-6, 8-11 and 15-18 stand rejected under 35 USC Section 102(b) as being anticipated by Frid-Nielsen, U.S. Patent Number 5,339,433 (hereinafter, “Frid-Nielsen”). Here, the Examiner equates Applicant’s claimed invention with the notion of symbol browsing from the code editor of a development environment, such as taught by Frid-Nielsen. However, as shown below, Applicant’s invention includes additional features which distinguish it from browsing.

Both browsing and Applicant’s Code Completion invention include techniques for discerning the relevance of a symbol (e.g., named variable or function) under examination, for instance, discerning a symbol’s data type and scope. However, Applicant’s invention extends this further concept further by using such information to help the user automatically complete input when creating program code (e.g., when entering source code text into a code editor of an integrated development environment).

The distinction is easily illustrated by way of example. Consider, for instance, a request to browse a *rect* variable (symbol) that has been created in one’s source code from a user-defined data type, *TRect*, which defines a rectangle data type. If the user requests browsing of the *rect* symbol, the user will be presented with graphical information illustrating

that *rect* is a variable created from the *TRect* data type. This is no doubt helpful to the user's understanding of the *rect* symbol in his or her source code.

However, what the user really would like is automated assistance with the task of actually inputting source code. Suppose, for instance, that the *TRect* data type is defined as follows (in the syntax of the C programming language):

```
typedef struct TRectTag
{
    LONG left;
    LONG top;
    LONG right;
    LONG bottom;
} TRect;
```

In the above example, the *TRect* data structure may be considered to define a rectangle by specifying four points (as long integers): *left*, *top*, *right*, *bottom*. Now suppose that the user would like automated help, when writing source code, with the task of setting the top/left coordinates to (0, 0) for the *rect* variable. Suppose that during code input, the user begins by typing the assignment statement for setting the left point to 0:

```
rect._
```

(where “_” indicates the current position of the screen cursor)

The user has to stop, though, as he or she does not recall the exact name of the first point of the first coordinate pair. Is it *LeftPoint*, or *LeftPt*, or *Left*, or *left*, or *X1*, or something else? (Note that the foregoing example is in the case-sensitive C programming language, where even *left* and *Left* are different.) Thus, what the user really would like help with is knowing what particular symbol(s) could be correctly typed in at this particular instance in time. With Applicant's invention providing the user an on-the-fly a list of valid data members -- *left*, *top*, *right*, *bottom* -- the user is easily able to discern that the correct symbol name for the first point is *left*. And, in fact, the user can complete the input

```
rect.left
```

without even typing “left”. Instead, the user simply selects “left” from the list of valid symbols (e.g., using the mouse or cursor keys).

More particularly, consider the example shown in Applicant’s specification at Fig. 5B, for the *myRect* symbol. There, the rectangle data type is defined in that environment to include not only *Left*, *Top*, *Right*, and *Bottom* integer data members but also defines coordinate pairs, *TopLeft* and *BottomRight*, thus giving the user the ability to specify points for a rectangle as coordinate pairs, such as (in Object Pascal syntax):

```
myRect.TopLeft := [0, 0]
```

Without Applicant’s automated Code Completion invention, the user might not have even realized that the *myRect* had additional data members: *TopLeft* and *BottomRight*.

Now consider the following assignment (in Object Pascal syntax):

```
iMyIntegerVariable := myRect._
```

(where “_” indicates the current position of the screen cursor)

Here, an integer variable (i.e., *iMyIntegerVariable*) is to receive the value of one of the data members of *myRect* (i.e., a data structure of type *TRect*). The Code Completion invention of the present invention is able to discern on-the-fly that only the integer data members of *myRect* (i.e., *Left*, *Top*, *Right*, and *Bottom*) are appropriate; the point data members (i.e., *TopLeft* and *BottomRight*) are not appropriate. Accordingly, only the integer data members are displayed to the user as appropriate for completing the expression. This level of functionality is not taught by symbol browsing. Quite simply, on-the-fly examination of an expression under construction for determining which data member(s) are appropriate for

completing the expression is not provided by symbol browsing, for instance as described by Frid-Nielsen (or elsewhere).

These distinctions are present as claim limitations in Applicant's claims.

Consider, for instance, the following limitation recited by claim 1:

determining input items which are suitable for input in the source code module at the current cursor position;
displaying to the user a list of said suitable input items; and
in response to selection by the user of a particular item from the list, automatically completing input at the current cursor position.

(emphasis added)

As shown above, the claim specifies selection of input items that are appropriate for completion of input at the current cursor position. (Independent claim 16 includes analogous limitations.) Symbol browsing technique, such as described by Frid-Nielsen, allows a user to determine the relationship of a symbol to other symbols in one's program (such as the relationship of an object variable to its corresponding class definition). This functionality, in effect, provides an "organizational chart" -- a hierarchical view -- for a given symbol, so that the user can determine its relevance relative to other symbols in his or her program. However, symbol browsing does not make any attempt whatsoever to actually complete a user source code statement or expression under construction. This requires the system to not only understand the symbol under examination and its neighboring symbols but also understand which symbol -- among the many available symbols -- would appropriately follow for correctly completing the current expression been inputted.

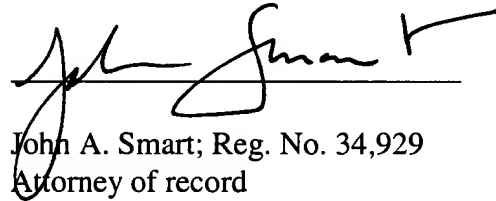
All told, symbol browsing as taught by Frid-Nielsen or elsewhere fails to teach or anticipate these claim limitations. Accordingly, it is respectfully requested that the Examiner reconsider the rejection under Section 102, particularly in light of the foregoing comments which further clarify the particular subject matter of the claims.

Conclusion

In view of the foregoing remarks and the amendment to the claims, it is believed that all claims are now in condition for allowance. Hence, it is respectfully requested that the application be passed to issue at an early date. If for any reason the Examiner feels that a telephone conference would in any way expedite prosecution of the subject application, the Examiner is invited to telephone the undersigned at (408) 395-8819.

Respectfully submitted,

Date: June 9, 2000



John A. Smart; Reg. No. 34,929
Attorney of record

Inprise Corporation
Legal Dept.
100 Enterprise Way
Scotts Valley, CA 95066
(408) 395-8819
(408) 490-2853 FAX

BORL/0170.00